Le tavole hash

È il modo con cui sono realizzati gli `HashSet`

Si usano quando serve accesso rapido sia in lettura che in scrittura su un insieme non ordinato

---

Principio base

Gli array hanno le caratteristiche che servono:

- lettura di un elemento: `vett[i]`
- scrittura di un elemento: `vett[i]=a`

Sono tutte e due operazioni dirette
(non richiedono nessuna ricerca)

Cerco di usare un vettore per rappresentare un insieme

Vettore caratteristico di un insieme

Un insieme di *numeri interi* che vanno da `0` a `100` si può rappresentare usando un vettore di cento elementi booleani

Se `v` è il vettore che rappresenta l'insieme, le operazioni si realizzano cosí:

lettura
: l'intero `x` si trova nell'insieme se e solo se `v[x]` è `true`

scrittura
: per inserire l'intero `x`, si fa `v[x]=true`
: per togliere, si fa `v[x]=false`

Insiemi generici

Limiti della rappresentazione dei vettori caratteristici:

- grandezza del vettore=numero di tutti gli elementi possibili
- si rappresentano solo interi

Si possono superare tutti e due grazie agli `hashCode`

Il metodo `hashCode`

È un metodo che ritorna un intero dato un oggetto

Dato un oggetto `o`, il suo intero corrispondente è `o.hashCode()`

Per il momento ci basta sapere questo

Modifica della rappresentazione

Al posto del vettore di booleani, usiamo un vettore di `Object`

- il codice hash di un oggetto identifica una posizione nel vettore
- se l'oggetto c'è, sta in quella posizione

Prima versione della tavola hash:

```
class Tavola {
  private Object t[];

  public Tavola() {
    t=new Object[...];
    // inizializzazione automatica a null
  }

  boolean add(Object o) {
    int c=o.hashCode();

    if(t[c]!=null)
      return false;

    t[c]=o;
    return true;
  }

  ...
```

Metodi `contains` e `remove`

```
class Tavola {
  ...

  boolean remove(Object o) {
    boolean p;
    int c=o.hashCode();

    p=(t[c]!=null);

    t[c]=null;

    return p;
  }

  boolean contains(Object o) {
    int c=o.hashCode();

    if(o==null)
      return false;

    if(t[c]==null)
      return false;

    return t[c].equals(o);
  }
}
```

Due problemi

- due oggetti possono avere lo stesso hashCode
- il vettore deve avere un elemento per ogni possibile numero intero
  Dato che si usano 32 bit per rappresentare un intero, servirebbe un vettore con `4294967296` elementi

Del primo problema ci occupiamo dopo

Dimensione del vettore

Scelgo una dimensione qualsiasi (per esempio, 100)

Quando l'hashCode vale `c`, uso `c%100` per indicizzare l'array

```
class Tavola {
  private final int size=100;
  private Object t[];

  public Tavola() {
    t=new Object[size];
  }

  boolean add(Object o) {
    int c=o.hashCode()%size;

    if(t[c]!=null)
      return false;

    t[c]=o;
    return true;
  }

  boolean remove(Object o) {
    boolean p;
    int c=o.hashCode()%size;

    p=(t[c]!=null);

    t[c]=null;

    return p;
  }

  boolean contains(Object o) {
    int c=o.hashCode()%size;

    if(o==null)
      return false;

    if(t[c]==null)
      return false;

    return t[c].equals(o);
  }
}
```

Stessa classe di prima con `c=o.hashCode()%size` al posto di `c=o.hashCode()`

Oggetti con lo stesso codice hash

Soluzione banale: quando vado a inserire un oggetto ma la sua posizione è già occupata, lo metto invece in una lista

Casella di un oggetto `o`: casella nella posizione `o.hashCode()%100`

dati
: un array e una lista

inserimento
: se la casella dell'oggetto è vuota, ci metto l'oggetto
: altrimenti, metto l'oggetto nella lista

ricerca
: se la casella dell'oggetto contiene `null`, l'oggetto non c'è
: se contiene un oggetto `equals`, ritorna `true`
: se contiene un oggetto che non è uguale, allora bisogna andare a vedere nella lista!

cancellazione
: stessa cosa; se l'oggetto sta nell'array, occorre anche spostare un eventuale oggetto dalla lista

---

Dati

Un vettore e una lista, più la dimensione del vettore

```
import java.util.*;

class Tavola {
  private final int size=100;
  private Object t[];
  private LinkedList l;

  public Tavola() {
    t=new Object[size];
    l=new LinkedList();
  }

  ...
}
```

La lista si chiama lista di trabocco

Assunzioni

Dato un oggetto `o`, la sua posizione naturale è:

> posizione naturale di un oggetto `o` = casella di indice `o.hashCode()%size` del vettore

Le assunzioni che faccio e rispetto sono:

1. un oggetto si trova preferibilmente nella sua posizione naturale
2. se la posizione naturale è occupata, allora l'oggetto sta da qualche altra parte

Nel nostro caso, "da qualche altra parte" significa nella lista

---

Inserimento

Facile: se la posizione naturale è libera, ci metto l'oggetto

Altrimenti, lo metto nella lista

Prima devo controllare che non sia presente

```
boolean add(Object o) {
  int c=o.hashCode()%size;

  if(t[c]!=null)
    if(l.contains(o))
      return false;
    else
      return l.add(o);

  t[c]=o;
  return true;
}
```

Ricerca

Devo considerare tutte e due le assunzioni:

- se l'oggetto lo trovo nella sua posizione naturale, allora c'è
- se nella sua posizione naturale c'è un altro oggetto, allora devo guardare la lista

```
boolean contains(Object o) {
  int c=o.hashCode()%size;

  if(t[c]==null)
    return false;

  if(t[c].equals(o))
    return true;

  return l.contains(o);
}
```

Cancellazione

Faccio prima una ricerca

Vedo se l'oggetto sta nella posizione naturale, e se è vuota guardo la lista

Se trovo l’oggetto nella posizione naturale, non basta cancellarlo

Se lo faccio, potrei avere un altro oggetto nella lista che ora non si trova nella sua posizione naturale anche se questa è libera!

Sarebbe un oggetto che non trovo quando faccio `contains`

Eliminazione dalla posizione naturale

Esempio: si supponga che gli oggetti `new Integer(50)` e `new Integer(150)` abbiano entrambi posizione naturale 50

Si inserisce prima `50` e poi `150`

Ora `50` sta nell’array e `150` nella lista

Si elimina `50` mettendo `null` nell’array

Se ora eseguo `contains(new Integer(150))`, mi ritorna `false`

Infatti, il metodo `contains` guarda nella posizione naturale, trova `null` e ritorna `false`

Eliminazione dalla posizione naturale

Due soluzioni:

- modifico `contains` in modo che cerca comunque nella lista
- modifico `remove`

La prima è inefficiente: quasi tutte le volte si deve andare a guardare nella lista

Usiamo la seconda soluzione

Eliminazione: modifica della remove

L’assunzione che viene usata da `contains` è che un elemento sta nella lista solo se la suo posizione naturale è occupata

Questo è lo stesso di: un elemento non può stare nella lista se la sua posizione naturale è libera

Quando si elimina un elemento dalla sua posizione naturale, basta prendere dalla lista uno qualsiasi degli altri elementi con quella posizione naturale (se esiste) e metterlo nella posizione naturale

Questo rende vera l’assunzione: se ci sono elementi nella lista, la loro posizione naturale è occupata

```
boolean remove(Object o) {
  boolean p;
  int c=o.hashCode()%size;

  if(o==null)
    return false;
```

```
    if(t[c]==null)
      return false;

    if(!t[c].equals(o))
      return l.remove(o);

// trovato oggetto: guarda la lista
    Iterator i=l.iterator();
    while(i.hasNext()) {
      Object b=i.next();
      if(b.hashCode()%size==c) {
        t[c]=b;
        i.remove();
        return true;
      }
    }

    t[c]=null;
    return true;
  }
```

Il metodo `hashCode`

Serve per trovare la posizione naturale di un oggetto in una tavola hash.

Vincolo: se due oggetti sono `equals`, devono avere lo stesso `hashCode`

La versione qui sotto rispetta questa specifica:

```
  public int hashCode() {
    return 0;
  }
```

Tutte le operazioni funzionano perchè la specifica è rispettata

---

Efficienza delle tavole hash

Se non vado mai nella lista di trabocco, ogni operazione ha costo uno (accesso al vettore)

Se tutti gli oggetti hanno lo stesso codice hash, allora tutti gli oggetti tranne uno stanno nella lista

L’efficienza delle operazioni è maggiore se riesco a non usare la lista di trabocco

Il metodo `hashCode` dovrebbe funzionare in modo tale da restituire interi possibilmente diversi per gli oggetti diversi che uso

Questo non può essere garantito; basta che due oggetti diversi che uso abbiano "spesso" codici diversi

---

Il metodo `hashCode`, rivisto

Vogliamo avere risultato diverso se i campi sono diversi

```
class Studente {
  String name;
  int anno;

  public int hashCode() {
    int res=0;

    res+=name.hashCode();
    res+=anno;

    return res;
  }
}
```

Prima versione: faccio la somma delle componenti intere e degli hashcode delle componenti oggetto

---

Problema della prima versione

Supponiamo di avere un insieme di `Point`

Non è improbabile che il punto `(1,2)` e `(2,1)` stiano nello stesso insieme

Hanno però lo stesso hashCode

Altro esempio: le persone con nome e cognome `Bruno, Marco` e `Marco, Bruno` hanno lo stesso hashCode se nome e cognome sono due componenti separate

---

Versione migliorata di `hashCode`

Soluzione: ogni componente viene moltiplicata per un valore diverso

```
class Studente {
  String name;
  int anno;

  public int hashCode() {
    int res=3;

    res=5*res+name.hashCode();
    res=5*res+anno;

    return res;
  }
}
```

L'ultima componente è moltiplicata per 1, la penultima per 5, la terz'ultima per 5*5, ecc.

Varianti delle tavole hash

I concetti comuni delle tavole hash sono:

1. per ogni oggetto, c'è una posizione naturale in un vettore
2. se questa posizione è occupata, l'oggetto sta "da qualche altra parte"

Ci sono versioni diverse in cui cambia il "qualche altra parte"

Versione senza lista

Se la posizione naturale di un oggetto è occupata, lo metto nella successiva posizione dell'array

Variante: invece della posizione successiva uso posizione+incremento, dove incremento è un numero primo rispetto alla dimensione dell'array

---

Senza lista: inserimento e cancellazione

inserimento
: nella posizione naturale se libera, altrimenti si cerca la prima posizione libera

ricerca
: nella posizione naturale; se c'è un oggetto ma non è uguale, si passa alla posizione successiva e si ripete ricorsivamente

---

Senza lista: cancellazione

È complicata

Esempio: (il numero e' la posizione naturale di un oggetto)

```
prima: {null null 2a 2b 2c 3d 3e null null}
```

Cancellare `2b`: non posso mettere solo `null` nella posizione 3, altrimenti otterrei:

```
dopo:  {null null 2a null 2c 3d 3e null null}
```

Se ora cerco `2c` oppure `3d`, ecc. non li trovo

Occorre spostare tutti gli elementi che seguono fino al primo elemento che è null

```
dopo:  {null null 2a 2c 3d 3e null null null}
```

In questi spostamenti, gli elementi non possono risalire prima della loro posizione naturale:

```
prima: {null null 2a 2b 2c 3d 3e 7f 3g}
non va:{null null 2a null 2c 3d 3e 7f null}
dopo:  {null null 2a 2c 3d 3e 3g 7f null}
```

Se non si rispettano queste regole, la cancellazione non è più coerente con la ricerca

---

Più liste di trabocco

Idea: per ogni elemento dell'array, ho una lista di trabocco

Ho quindi un array di oggetti e un array di liste;
i due array hanno la stessa dimensione

L’elemento sta nella posizione naturale oppure nella lista di trabocco della posizione naturale

---

Buckets

Realizzo solo l’array di liste

Il principio è che l’accesso al primo elemento di lista è comunque facile come accedere a un elemento di un array

Ognuna delle liste si chiama *bucket*

Realizzazione

```
import java.util.*;

class Tavola {
  private final int size=100;
  private LinkedList l[];

  public Tavola() {
    int i;

    l=new LinkedList[size];
    for(i=0; i<l.length; i++)
      l[i]=new LinkedList();

  }

  boolean add(Object o) {
    int c=o.hashCode()%size;

    if(l[c].contains(o))
      return false;

    return l[c].add(o);
  }

  boolean contains(Object o) {
    int c=o.hashCode()%size;

    return l[c].contains(o);
  }

  boolean remove(Object o) {
    int c=o.hashCode()%size;

    return l[c].remove(o);
  }
}
```

Realizzazione: nota

Si poteva anche lasciare l’array di liste vuoto

La lista `l[c]` veniva creato solo quando si andava a inserire un elemento che aveva posizione naturale `c`

Bisogna fare un controllo `l[c]==null` sia su `remove` che su `contains`